Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 06.02.2007: Schlußbemerkungen

Christoph Lüth

Inhalt der Vorlesung

- Organisatorisches
- Noch ein paar Haskell-Döntjes:
  - Concurrent Haskell
  - HaXml — XML in Haskell
- Rückblick über die Vorlesung
- Ausblick

Der studienbegleitende Leistungsnachweis

- Bitte Scheinvordruck ausfüllen.
  - Siehe Anleitung.
  - Erhältlich vor FB3-Verwaltung (MZH Ebene 7)
  - Nur wer ausgefüllten Scheinvordruck abgibt, erhält auch einen.
- Bei Sylvie Rauer (Cartesium 2.046) oder mir (Cartesium 2.045) oder Tutor abgeben (oder zum Fachgespräch mitbringen)

Scheinvordruck

Universität Bremen
- Fachbereich 3 -[1]

AUSBILDUNGSBEGLEITENDE LEISTUNGSKONTROLLE[2]

Von der Studentin/vom Studenten auszufüllen:[3]

Name: NAME Matr. Nr. 21864223

Titel der Lehrveranstaltung: Praktische Informatik 3

Veranstaltungskennziffer 03-05-G-700.03 SS/WS 06/07 Wochenstunden: 4 (ECTS: 6)

Anerkannt für: Informatik-Diplom / Bachelor

Nicht von der Studentin/vom Studenten auszufüllen:

Die/der obengenannte Studentin/Student hat an der vorerwähnten Lehrveranstaltung erfolgreich teilgenommen und folgende Leistung erbracht:

Einzelleistung: ☐ Gruppenleistung: ☒ (Zutreffendes ankreuzen)

Inhalt und Form der Leistung:

Erfolgreiche Bearbeitung der Übungsblätter

Beurteilung:

Das Fachgespräch

- Dient zur Überprüfung der Individualität der Leistung
  - Insbesondere: Teilnahme an Bearbeitung der Übungsblätter
  - Keine Prüfung.
- Dauer: ca. 10 Min; einzeln, auf Wunsch mit Beisitzer
- Inhalt: Übungsblätter
- Bearbeitete Übungsblätter mitbringen — es werden zwei Aufgaben besprochen, die erste könnt Ihr Euch aussuchen
- Termine:
  - Di 20.02 und Mi 21.02, Anmeldung auf Webseite
  - . . . oder nach Vereinbarung

Fallbeispiel: XML in Haskell

- Vorteile von strenger Typisierung, algebraische Datentypen.
- XML:
  - Eine Notation für polynomiale Datentypen mit viel < und >
  - Eine Ansammlung darauf aufbauender Techniken und Sprachen

Fallbeispiel: XML in Haskell

- Vorteile von strenger Typisierung, algebraische Datentypen.
- XML:
  - Eine Notation für polynomiale Datentypen mit viel < und >
  - Eine Ansammlung darauf aufbauender Techniken und Sprachen
- XML hat mehrere Schichten:
  - Basis: Definition von semantisch strukturierten Dokumenten
  - Präsentation von strukturierten Dokumenten
  - EinheitlicheDefinitionen und Dialekte

XML — Eine Einführung

- Frei definierbare Elemente und Attribute
- Elemente und Attribute werden in DTD definiert
  - Heutzutage: Schemas, RelaxNG
  - Entspricht einer Grammatik
- Beispiel: Ein Buch habe
  - Autor(en) — mindestens einen `[String]` (nichtleer)
  - Titel `String`
  - evtl. eine Zusammenfassung `[String]`
  - ISBN-Nummer `String`

Beispiel: RelaxNG Schema für Bücher

```
start= element book { attribute isbn { text }
                    , author+, title, abstract? }
author   = element author {text}
title    = element title {text}
abstract = element abstract {para*}
para     = element paragraph {text}
```

Beispiel: DTD für Bücher

```
<?xml encoding="UTF-8"?>
<!ELEMENT book (author+,title,abstract?)>
<!ATTLIST book
  isbn CDATA #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT abstract (paragraph)*>
<!ELEMENT paragraph (#PCDATA)>
```

- `book`, ..., `paragraph`: Elemente
- `isbn`: Attribut

Ein Beispiel-Buch

```
<?xml version='1.0'?>
<!DOCTYPE book SYSTEM "book.dtd">
<book isbn="3540900357">
  <author>Saunders MacLane</author>
  <title>Categories for the Working Mathematician</title>
  <abstract>
    <paragraph>The introduction to category theory
    by one of its principal inventors.</paragraph>
 </abstract>
</book>
```

HaXml

- HaXML: getypte Einbettung in Haskell
  - DTD $\longrightarrow$ algebraischen Datentyp
  - Klasse `XmlContent`
  - Funktionen

```
readXml :: XmlContent a => String -> Maybe a
showXml :: XmlContent a => a -> String
```

  - Jedes Element einen Typ, Instanz von `XmlContent`
- Übersetzung der DTD: `DtdToHaskell book.dtd Book.hs`

Generiertes Haskell

```
module Book where
data Book = Book Book_Attrs (List1 Author) Title (Maybe Abstra
          deriving (Eq,Show)
data Book_Attrs = Book_Attrs
    { bookIsbn :: String
    } deriving (Eq,Show)
newtype Author = Author String  deriving (Eq,Show)
newtype Title = Title String  deriving (Eq,Show)
newtype Abstract = Abstract [Paragraph]  deriving (Eq,Show)
newtype Paragraph = Paragraph String  deriving (Eq,Show)

instance XmlContent Book where
    ...
```

Vorteile

- Einfache, getypte Manipulation von XML-Dokumenten
- Beispiel formatierte Ausgabe (Zeigen):

```
prt :: Book-> String
prt (Book (Book_Attrs {bookIsbn=i}) (NonEmpty as) (Title t)
    authors++ ": "++ t++ "\n"++ abstr++ "ISBN: "++ i where
    authors = if length nms > 3 then  head nms ++ " et al"
              else concat (intersperse ", " nms)
    nms    = map (\ (Author a)-> a) as
    abstr = case abs of
              Just (Abstract a)-> unlines (map
                                  (\ (Paragraph p)-> p) a) ++ "\n"
              Nothing -> ""
```

Noch ein Beispiel: Suche

- ... in Bücherei nach Stichwörtern in Titel oder Zusammenfassung:
- Erweiterte DTD `library.dtd`

```
<!ELEMENT library (book*)>
```

- Damit generierter Typ

```
newtype Library = Library [Book] deriving (Eq,Show)
```

- Hilfsfunktion: Liste aller Wörter aus Titel und Zusammenfassung

```
getWords :: Book-> [String]
getWords (Book _ _ (Title t) Nothing) = words t
getWords (Book _ _ (Title t) (Just (Abstract a))) =
  words t ++ (concat (map (\ (Paragraph p)-> words p) a))
```

Suche

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem key. getWords) bs
```

- Nachteil: Groß/Kleinschreibung relevant

Suche

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem key. getWords) bs
```

- Nachteil: Groß/Kleinschreibung relevant
- Deshalb alles in Kleinbuchstaben wandeln:

```
query :: Library-> String-> [Book]
query (Library bs) key =
  filter (elem (map toLower key).
               (map (map toLower).getWords)) bs
```

Verbesserung: Suche mit regulären Ausdrücken

- Nutzt Regex-Bücherei des GHC:

```
import Text.Regex
data Regex -- abstrakt
mkRegex :: String -> Regex -- übersetzt regex
matchRegex :: Regex -> String -> Maybe [String]
```

- Damit neue Version von `query` (Zeigen):

```
query' (Library bs) ex =
  filter (any (isJust.matchRegex (mkRegex ex)).
          getWords) bs
```

Zusammenfassung HaXML

- Transformation in Haskell einfacher, typsicherer &c. als style sheets, XSLT, &c
- Durch XML beschriebene Datentypen sind algebraisch
- Nützlich zum Datenaustausch — nahtlose Typsicherheit, e.g.

$$\texttt{Java} \longleftrightarrow \texttt{Haskell} \longleftrightarrow \texttt{C++}$$

  - z.B. JAXB (`http://java.sun.com/webservices/jaxb/`)
  - Leichtgewichtige Komponenten

Concurrent Haskell

- Threads in Haskell:

```
forkIO :: IO () -> IO ThreadID
killThread :: ThreadID -> IO ()
```

- Zusätzliche Primitive zur Synchronisation
- Erleichtert Programmierung reaktiver Systeme
  - Benutzerschnittstellen, Netzapplikationen, ...
- hugs: kooperativ, ghc: präemptiv

Beispiel: Einfache Nebenläufigkeit

Beispiel: Zeigen

```
module Main where

import Control.Concurrent

write :: Char -> IO ()
write c = putChar c >> write c

main :: IO ()
main = forkIO (write 'X') >> write '.'
```

Nebenläufigkeit in Haskell

- Synchronisationsmechanismen: `MVar a`
- Können nebenläufig gelesen/geschrieben werden
- Nach dem Schreiben voll, nach dem Lesen leer
- Blockiert wenn
  - aus leerer `MVar` lesen
  - in volle `MVar` schreiben
- Seiteneffekt
- Damit Schlangen (Queues), Kanäle, Semaphoren, ...

Der Haskell Web Server

- Ein RFC-2616 konformanter Webserver (Peyton Jones, Marlow 2000)
- Beispiel für ein
  - nebenläufiges,
  - robustes,
  - fehlertolerantes,
  - performantes System.
- Umfang: ca. 1500 LOC, „written with minimal effort“
- Performance: ca. 100 req/s bis 700 req
- `http://www.haskell.org/~simonmar/papers/web-server.ps.gz`

Grundlagen der funktionalen Programmierung

- Definition von Funktionen durch rekursive Gleichungen
- Auswertung durch Reduktion von Ausdrücken
- Typisierung und Polymorphie
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Beweis durch strukturelle und Fixpunktinduktion

Fortgeschrittene Features

- Modellierung von Zustandsabhängigkeit durch `IO`
- Überladene Funktionen durch Typklassen
- Unendliche Datenstrukturen und verzögerte Auswertung
- Beispiele:
  - Parserkombinatoren
  - Grafikprogrammierung
  - Animation

Was war wichtig?

- Funktionen als Programme: alle Abhängigkeiten explizit
- Algebraische Datentypen
- Abstraktion und abstrakte Datentypen

Zusammenfassung Haskell

Stärken:

- Abstraktion durch
  - Polymorphie und Typsystem
  - algebraische Datentypen
  - Funktionen höherer Ordnung
- Flexible Syntax
- Haskell als Meta-Sprache
- Ausgereifter Compiler
- Viele Büchereien

Schwächen:

- Komplexität
- Dokumentation
  - z.B. im Vergleich zu Java's APIs
- Büchereien
- Noch viel im Fluß
  - Tools ändern sich
  - Zum Beispiel FFI/HGL
- Entwicklungsumgebungen

Andere Funktionale Sprachen

- Standard ML (SML):
  - Streng typisiert, strikte Auswertung
  - Formal definierte Semantik
  - Drei aktiv unterstütze Compiler
  - `http://www.standardml.org/`
- Caml, O'Caml:
  - Streng typisiert, strikte Auswertung
  - Hocheffizienter Compiler, byte code & native
  - Nur ein Compiler (O'Caml)
  - `http://caml.inria.fr/`

Andere Funktionale Sprachen

- LISP & Scheme
  - Ungetypt/schwach getypt
  - Seiteneffekte
  - Viele effiziente Compiler, aber viele Dialekte
  - Auch industriell verwendet

Funktionale Programmierung in der Industrie

- Erlang
  - schwach typisiert, nebenläufig, strikt
  - Fa. Ericsson — Telekom-Anwendungen
- FL
  - ML-artige Sprache
  - Chip-Verifikation der Fa. Intel
- Galois Connections
  - Hochqualitätssoftware in Haskell
  - Hochsicherheitswebserver, Cryptoalgorithmen

Perspektiven

- Funktionale Programmierung in 10 Jahren?
- Anwendungen:
  - Integration von XML, DBS (X#/Xen, Microsoft)
  - Integration in Rahmenwerke (F# & .Net, Microsoft)
  - Eingebettete domänenspezifische Sprachen
- Forschung:
  - Ausdrucksstärkere Typsysteme
  - für effiziente Implementierungen
  - und eingebaute Korrektheit (Typ als Spezifikation)
  - Parallelität?

Was lernt uns funktionale Programmierung?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen

Was lernt uns funktionale Programmierung?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen
- Konzentration auf wesentliche Elemente moderner Programmierung:
  - Typisierung und Spezifikation
  - Datenabstraktion
  - Modularisierung und Dekomposition

Was lernt uns funktionale Programmierung?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen
- Konzentration auf wesentliche Elemente moderner Programmierung:
  - Typisierung und Spezifikation
  - Datenabstraktion
  - Modularisierung und Dekomposition

- Blick über den Tellerrand — Blick in die Zukunft
  - Studium $\neq$ Programmierkurs— was kommt in 10 Jahren?

Hilfe!

- Haskell: primäre Entwicklungssprache am DFKI, FG SKS
  - Sicherheit in der Robotik: `http://www.dfki.de/sks/sams`
- Wir suchen studentische Hilfskräfte
  - für diese Projekte
- Wir bieten:
  - Angenehmes Arbeitsumfeld
  - Interessante Tätigkeit
- Wir suchen Tutoren für PI3
  - im WS 05/06 — meldet Euch bei Berthold Hoffmann!

Tschüß!